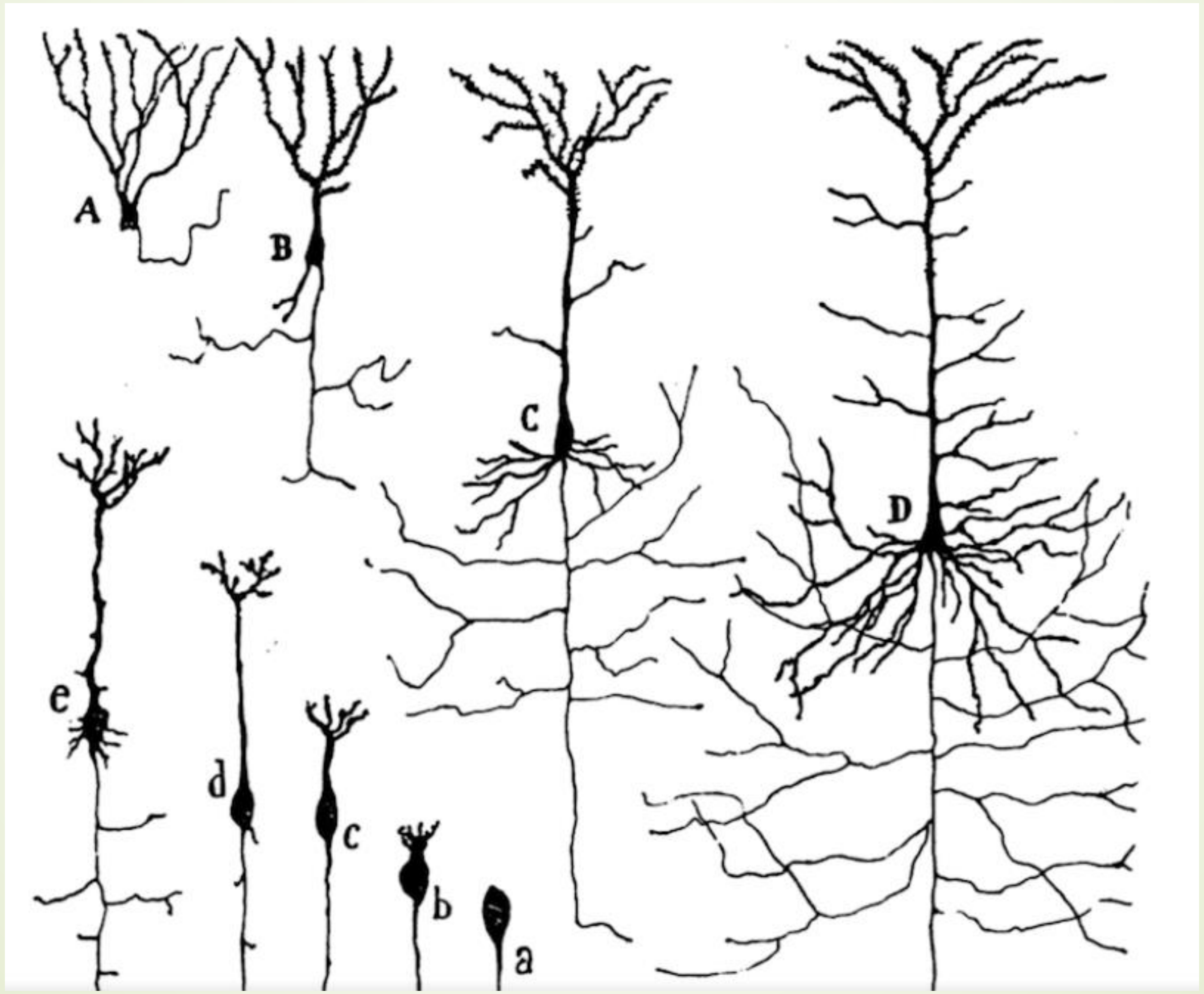
Deep Learning with CUDA Neurons and Networks

Tomasz Szumlak

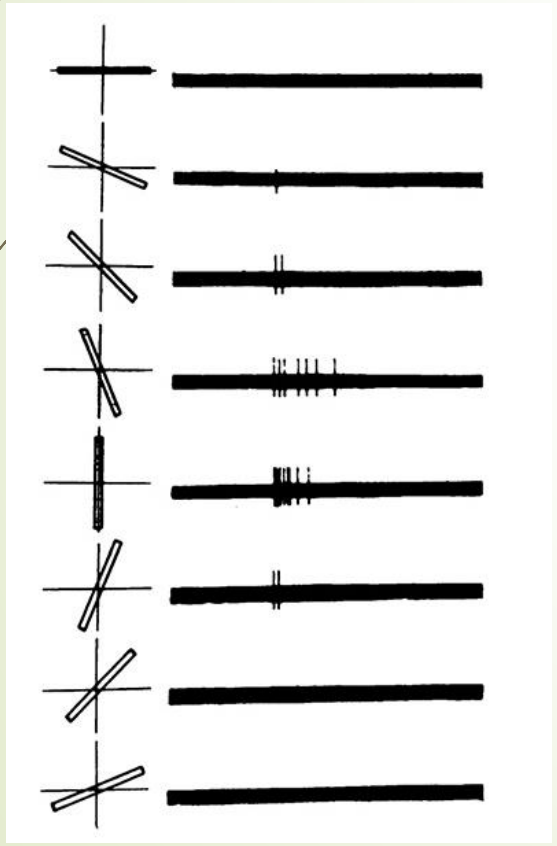
WFiIS AGH
07-13/03/2024, Kraków

ANN – inspiration from biology



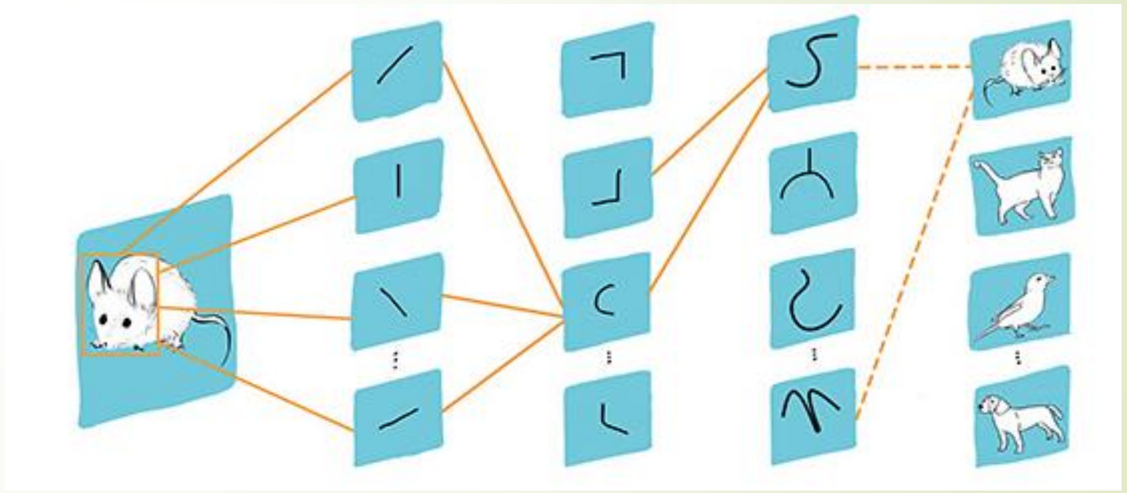
CV – new brave world with CNN

Hubel, D. H., & Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology, 148*, 574–91.

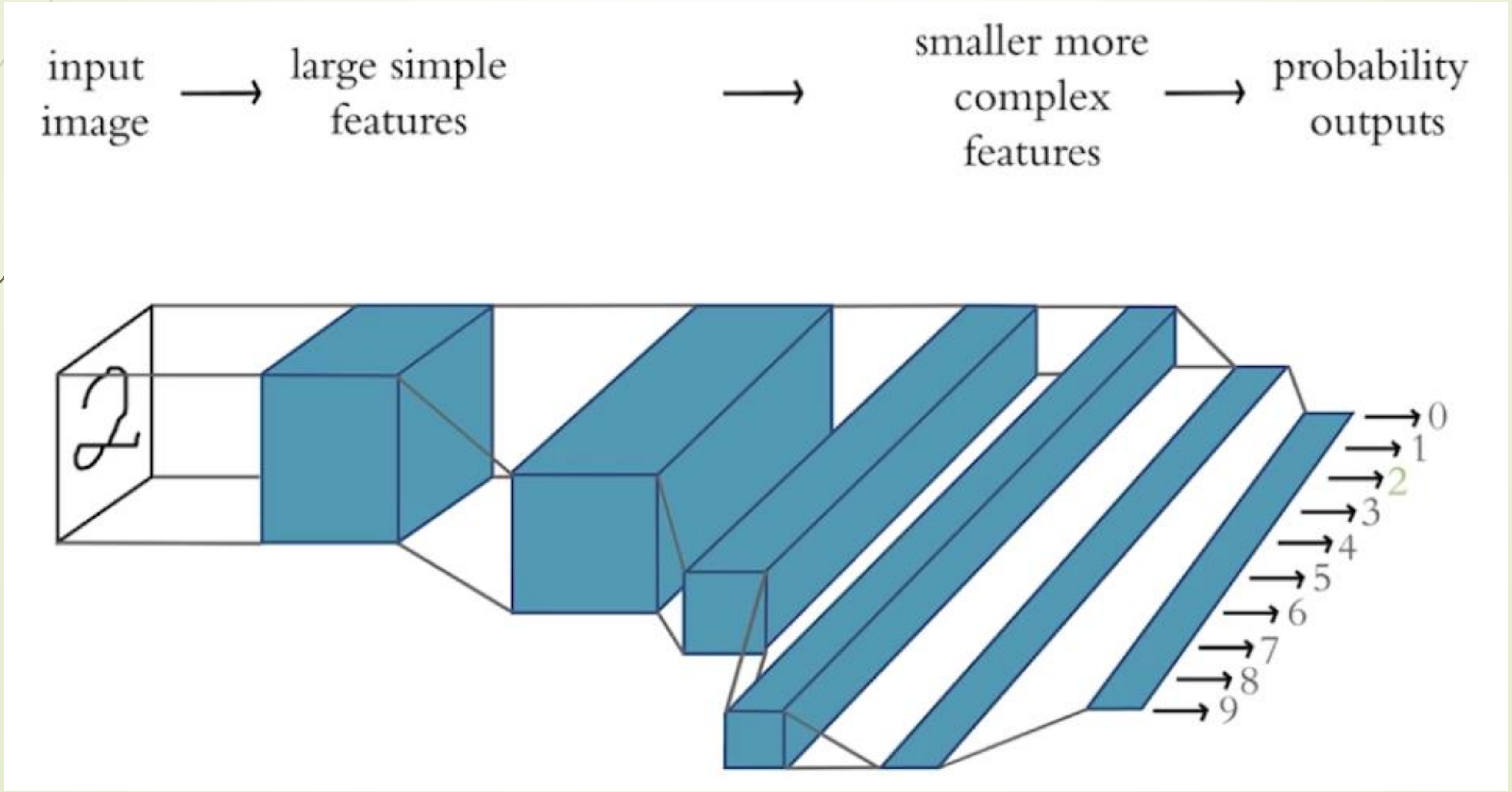


Simple neurons

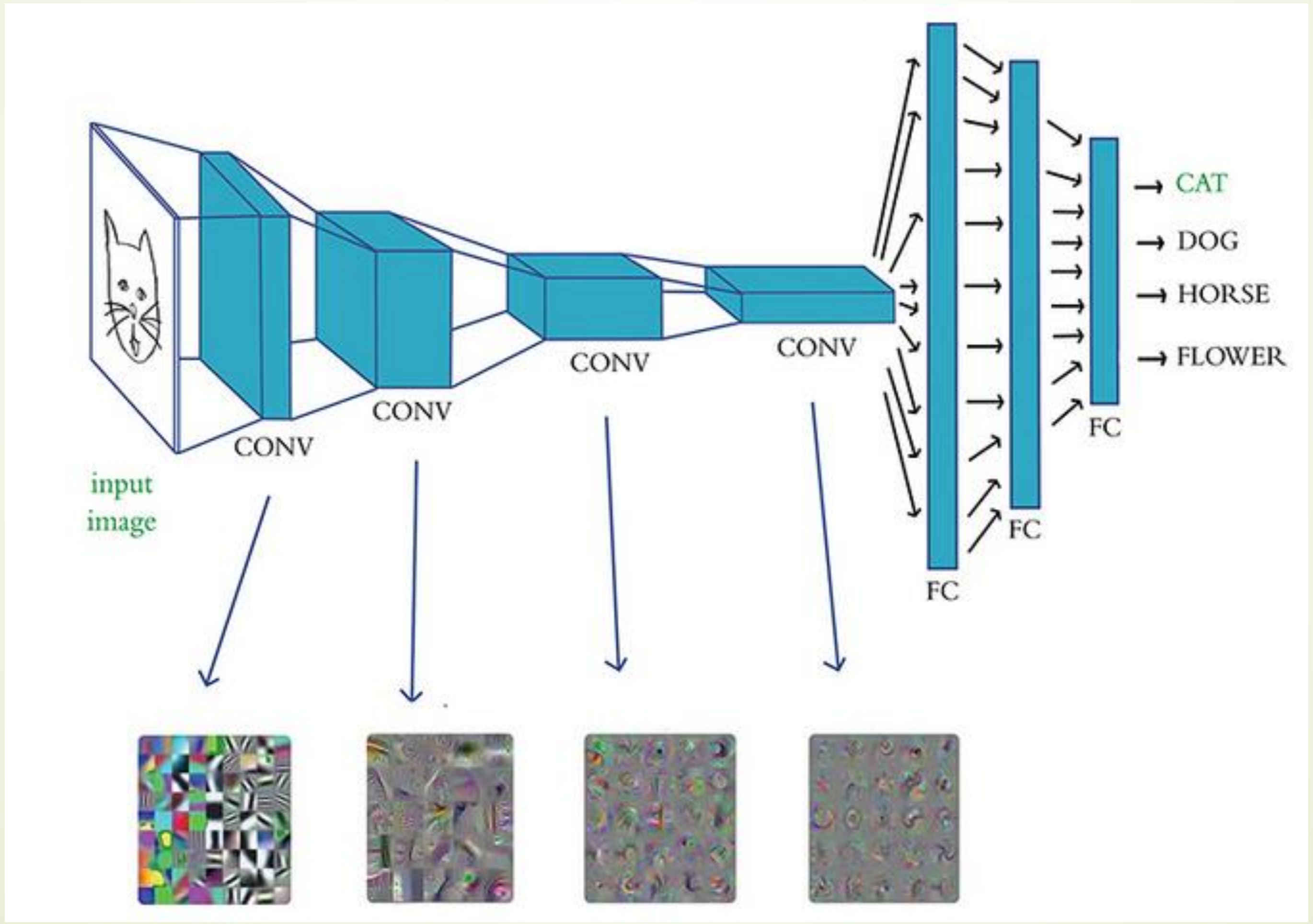
Complex neurons



LeNET-5 multi-class classification

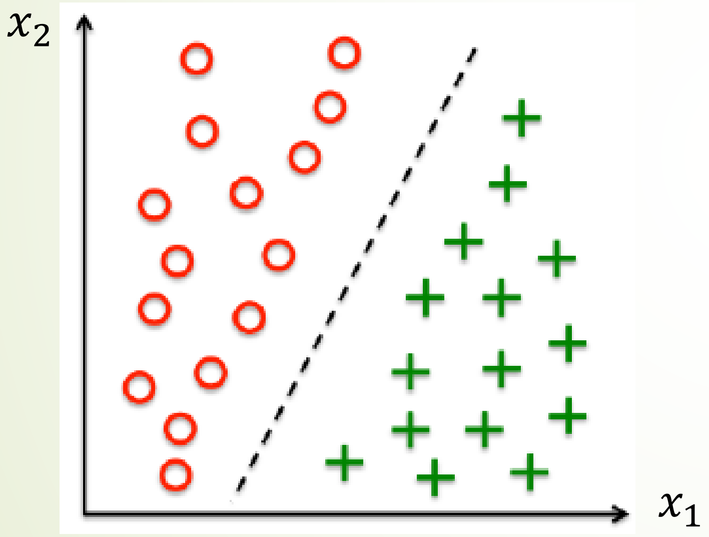


Deep colvolutional model



Binary classification

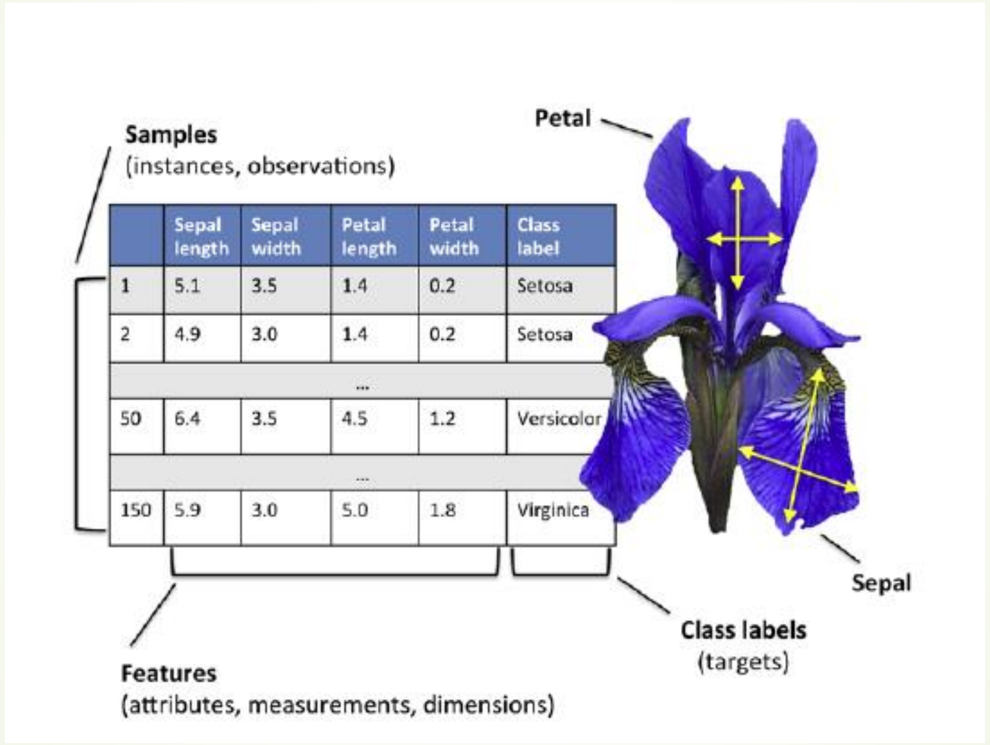
- The idea of a binary classification can be understood using the following example: say, we have given 30 training samples – half of them is **negative** (noise) and half positive (signal)



- 2D data set – each data instance has two values (x_1, x_2) associated with it
- Using them separately is going to yield poor results!
- Try to imagine we project the data on the respective axes

- Our algorithm must learn a rule to separate these two classes and classify a new instance into one of these classes given values x_1, x_2
- This rule is also called **decision boundary** (black dashed line)

Terminology


Samples
(instances, observations)
Petal
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

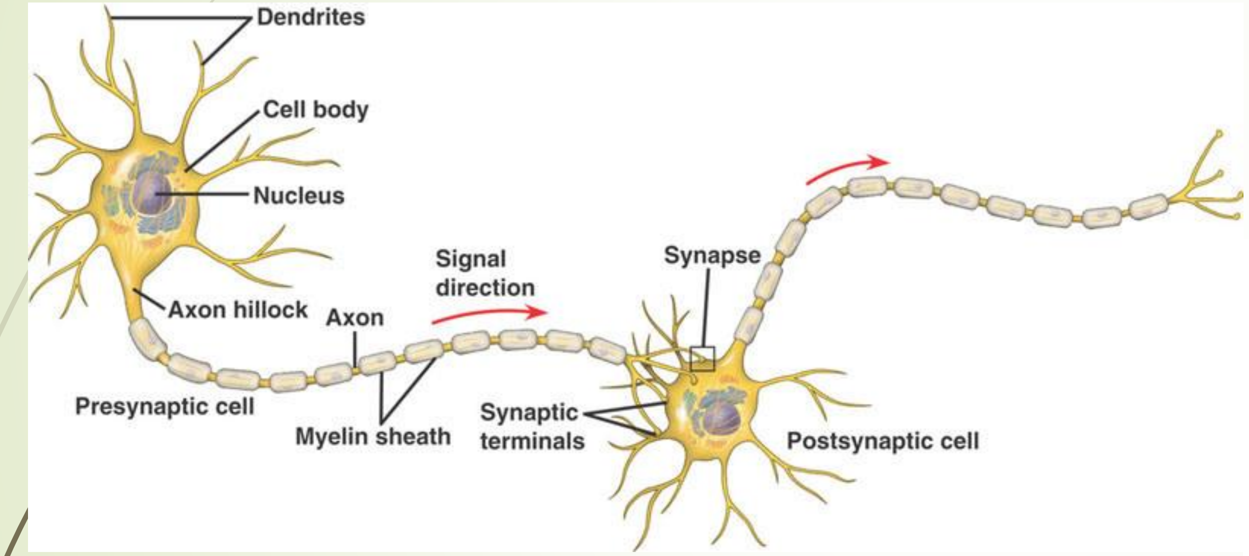
$$x_j = \begin{pmatrix} x_j^{(1)} \\ . \\ . \\ . \\ x_j^{(m)} \end{pmatrix}$$

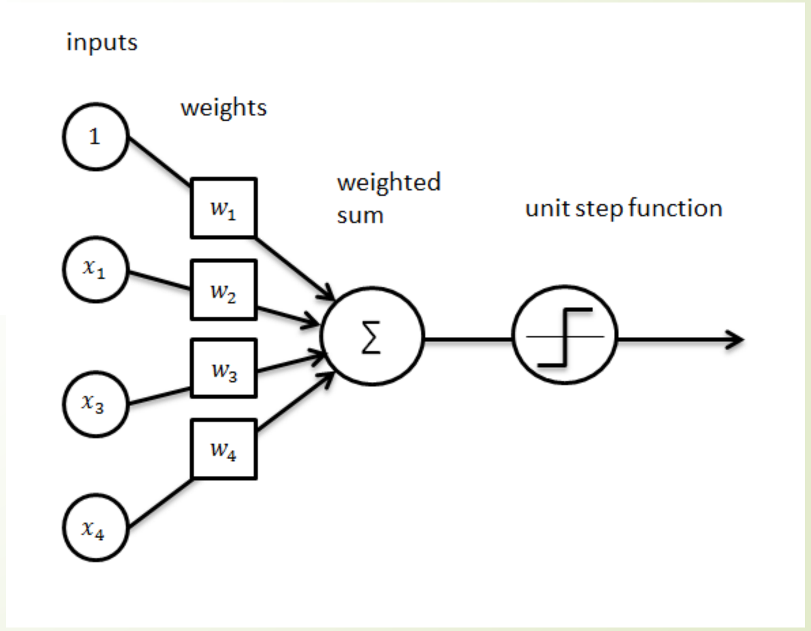
$x^{(i)} = \left(x_1^{(i)}, \dots, x_n^{(i)}\right)$ - one instance

A deeper look…

- The whole story began in **1943** with McCullock-Pitts **neuron model**
- They described a **nerve cell** using a simple **binary logic gate** with binary output(s) – such **perceptron** (or artificial neuron) accepts multiple input signals, **integrates** (combines) them and if signal exceeds a certain threshold the perceptron is able to produce an **output signal**
  - Using the biological language, the signal arrives via **dendrites**, is then processed by a **cell body** and the output is **propagated via axon**
  - Axon can have **many terminals** connecting the perceptron with others
- Next in 1957 **Rosenblatt** came with an brilliant idea on how to efficiently **train** such perceptrons
- With this learning rule it is possible to determine automatically the best weights that decide if the perceptron fires or not

Perceptron model





Rosenblatt approach

- With Rosenblatt method we use the supervised learning to train a set of weights
- They are then multiplied with the input data (features) and based on the result a decision is made: fire/not fire
- Trained perceptron algorithm can be subsequently used to make a decision regarding a sample membership (classification)
- In order to provide a bit more formal description, let's assume that we are dealing with a binary classification task with two classes, we then call one a **positive** $(+1)$ and the other a **negative** (-1)
- In order to define our perceptron behaviour we need to define an **activation function**: $\phi(z)$

$$\phi(z) = \begin{cases} +1 \ if \ z \geq \theta \\ -1 \ if \ z < \theta \end{cases}$$

Predefined threshold

Some math… (I)

- The activation function, defined in the previous slide as the Heaviside **step function** (NOTE! This is not a unique choice) takes a linear combination of given input values and a weight vector

$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{bmatrix}, \qquad \vec{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_k^{(i)} \end{bmatrix}$$

- Using the two vectors we can calculate the **net input** z:

$$z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots + w_k x_k^{(i)} = \sum_{j=1}^{j=k} w_j x_j^{(i)} = \vec{w}^T \vec{x}^{(i)}$$

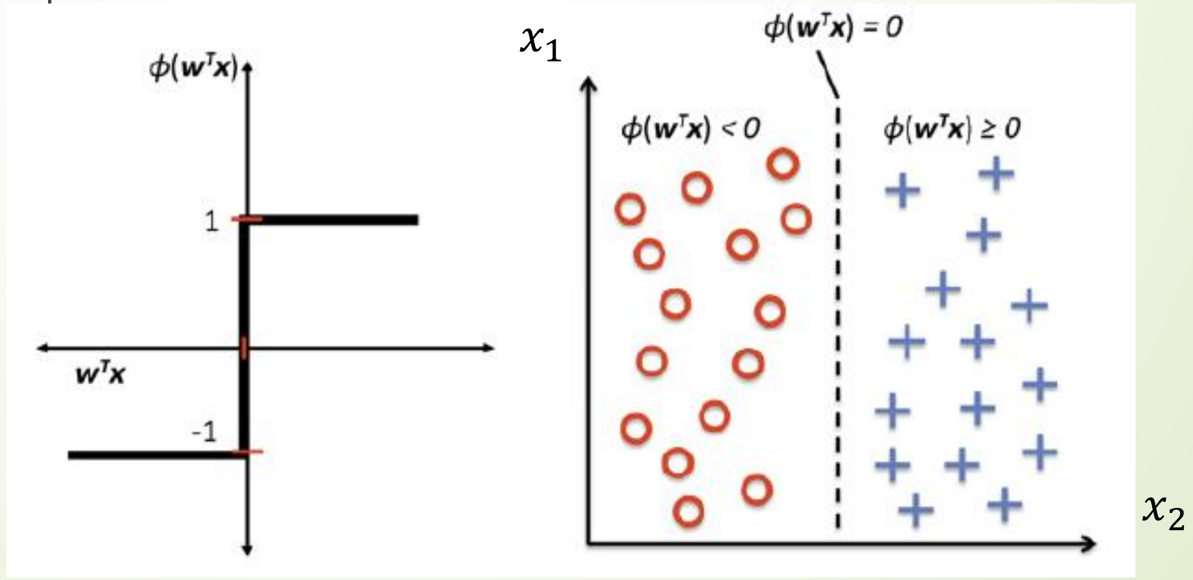
- Usually, we also move the threshold to the left side to facilitate the notation:

$$z'^{(i)} = w_0 x_0^{(i)} + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots + w_k x_k^{(i)}, w_0 = -\theta, x_0^{(i)} = 1$$

$$\boxed{\sum_{j=1}^{j=k} w_j x_j^{(i)} + b = \vec{w}^T \vec{x}^{(i)} + b = \vec{w}^T \vec{x}^{(i)} - \theta = z'^{(i)}}$$

Some math… (II)

- Here, a critical part is to understand two things:
  - How the **input information** (which may have many components) is **translated** into a binary information $(+1/-1)$?
  - How it is used to make the classification?
- Note, that the formulas we wrote in the last slide are identical to the vector dot product – we reduce the space dimension!



Adapted from „Python Machine Learning", S. Raschka

The algorithm (I)

- First we reduce the input data and mimic the behaviour of a single neuron (just like in the brain): fire/not fire
- The perceptron algorithm, then goes like that:
  - **Initialise the weights vector to $\mathbf{0}$ or „something small”**
  - **For each training data sample $\vec{x}^{(i)}$ do:**
    - **Get the output value (class label) $\tilde{y}^{(i)}$, using the unit step function**
    - **Update the weights accordingly (update concerns all the weights in one go)**
- We can write

$$w_j = w_j + \Delta w_j$$

$$\boldsymbol{\Delta w_j = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_j^{(i)}}$$

- The second formula is called **perceptron learning rule**, and the $\boldsymbol{\eta}$ is called the learning rate (just a number between 0 and 1)

The algorithm (II)

- To update the weight vector we need to know the true class label $y^{(i)}$ and calculate the predicted one: $\tilde{y}^{(i)}$
- All the weights are updated at once, we do not re-compute the class label before all Δw_j are updated
- Looking at our 2D example problem (see slide 7) we would write:

$$\Delta w_0 = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right)$$

$$\Delta w_1 = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_1^{(i)}$$

$$\Delta w_2 = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_2^{(i)}$$

- Before we devour our keyboards in order to implement the perceptron experiment, let's make the following „gedanken" experiment to get the better feeling about it

The algorithm (III)

- So, let's check first what happens to the weights if we predict the class label correctly/incorrectly:

$$\Delta w_j = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_j^{(i)} = \eta \cdot \left(-1^{(i)} - \left(-1^{(i)}\right)\right) \cdot x_j^{(i)} = 0$$

$$\Delta w_j = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_j^{(i)} = \eta \cdot \left(+1^{(i)} - \left(+1^{(i)}\right)\right) \cdot x_j^{(i)} = 0$$

- If the prediction is wrong, however, we have:

$$\Delta w_j = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_j^{(i)} = \eta \cdot \left(1^{(i)} - \left(-1^{(i)}\right)\right) \cdot x_j^{(i)} = 2 \cdot \eta \cdot x_j^{(i)}$$

$$\Delta w_j = \eta \cdot \left(y^{(i)} - \tilde{y}^{(i)}\right) \cdot x_j^{(i)} = \eta \cdot \left(-1^{(i)} - \left(1^{(i)}\right)\right) \cdot x_j^{(i)} = -2 \cdot \eta \cdot x_j^{(i)}$$

- So, the algorithm will try to push weights toward the value of the target class label
- And what about the value of the sample components?

The algorithm (IV)

- Consider the following:

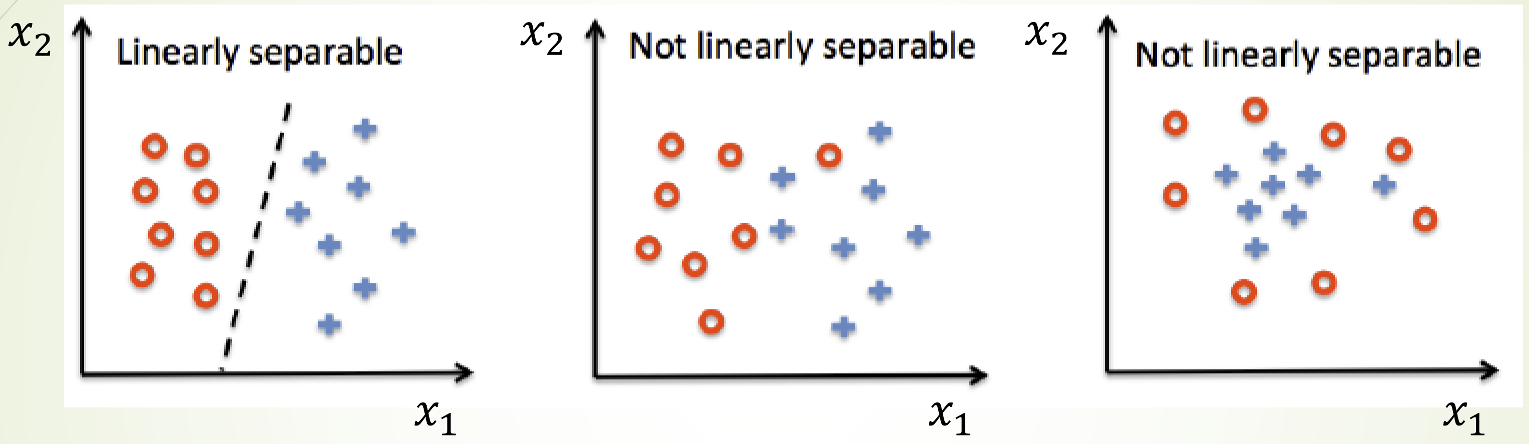
$$y^{(i)} = +1, \tilde{y}^{(i)} = -1, \eta = 1$$

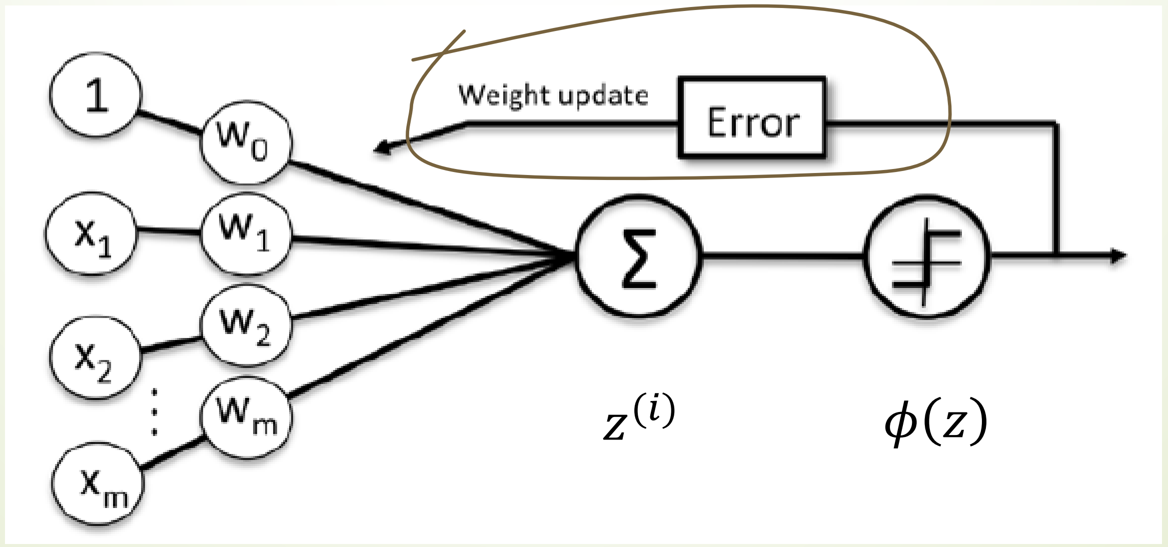
- So, the true label is $+1$, however we predicted -1, with the fixed learning rate it turns out that the weight update is proportional to the value of the variable

$$\Delta w_j = \eta \cdot \left(1^{(i)} - \left(-1^{(i)}\right)\right) \cdot x_j^{(i)} = 1 \cdot 2 \cdot x_j^{(i)}$$

- Next time we encounter such event the weight will be more positive (or negative, depending on the value of $x_j^{(i)}$)
- The convergence of the perceptron algorithm is only possible when the two **classes are linearly separable** (and with a small learning rate)
- If the above is not true we need to make a decision on a maximum **number of epochs** (or how many times we go over a training data) and a **maximum number of acceptable errors** in classification

The overview





Make some compromise

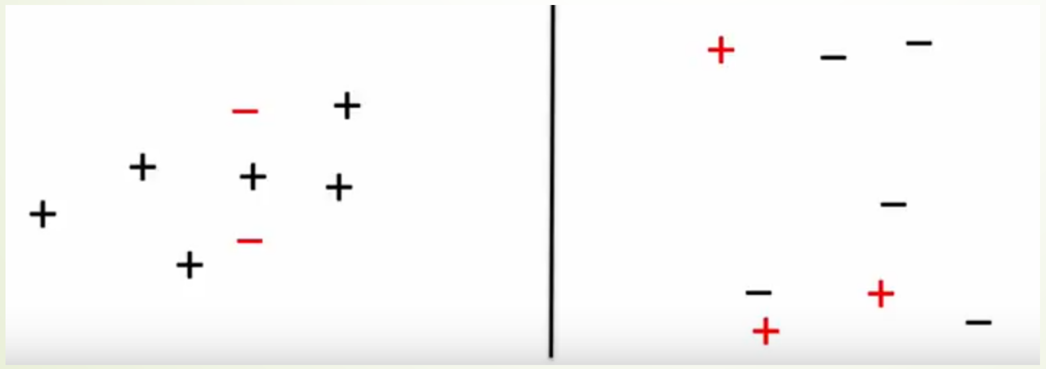
Loss function (I)

- In practice we need to have a very good handle on the performance of our model
- Or, in other words we need to have means to penalise the model if it performs poorly and reward if it does good
- We could, for instance, just count the number of good and bad decisions and calculate the rates. Imagine the situation below

$\mathcal{M}(\vec{x}) = 0$

$\mathcal{M}(\vec{x}) > 0$ $\mathcal{M}(\vec{x}) < 0$

Red points are misclassified



$$\mathcal{L}(y_i, \mathcal{M}(\vec{x}_i)) = \frac{1}{n}\sum_i [y_i \neq sign(\mathcal{M}(\vec{x}_i))]$$

Loss function (II)

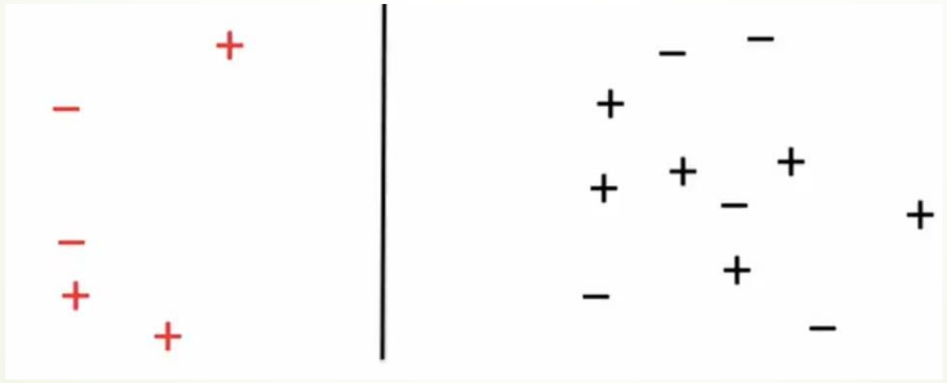
- Let's create „an universal" formula for the loss function, for that imagine we moved all the correctly classified points on one side and the misclassified on the other (note, we are changing the meaning of the plot from the last slide!

The opposite signs

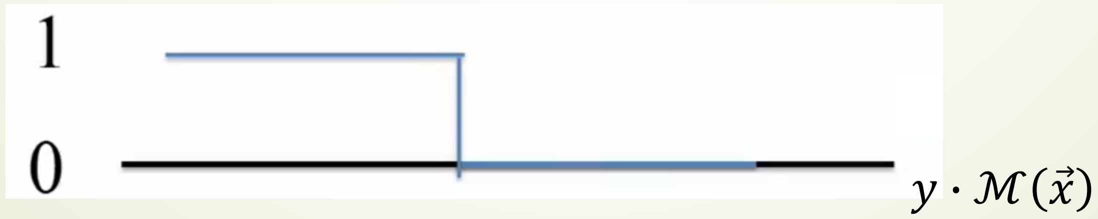
$y \cdot \mathcal{M}(\vec{x}) < 0$

$y \cdot \mathcal{M}(\vec{x}) > 0$

The same signs



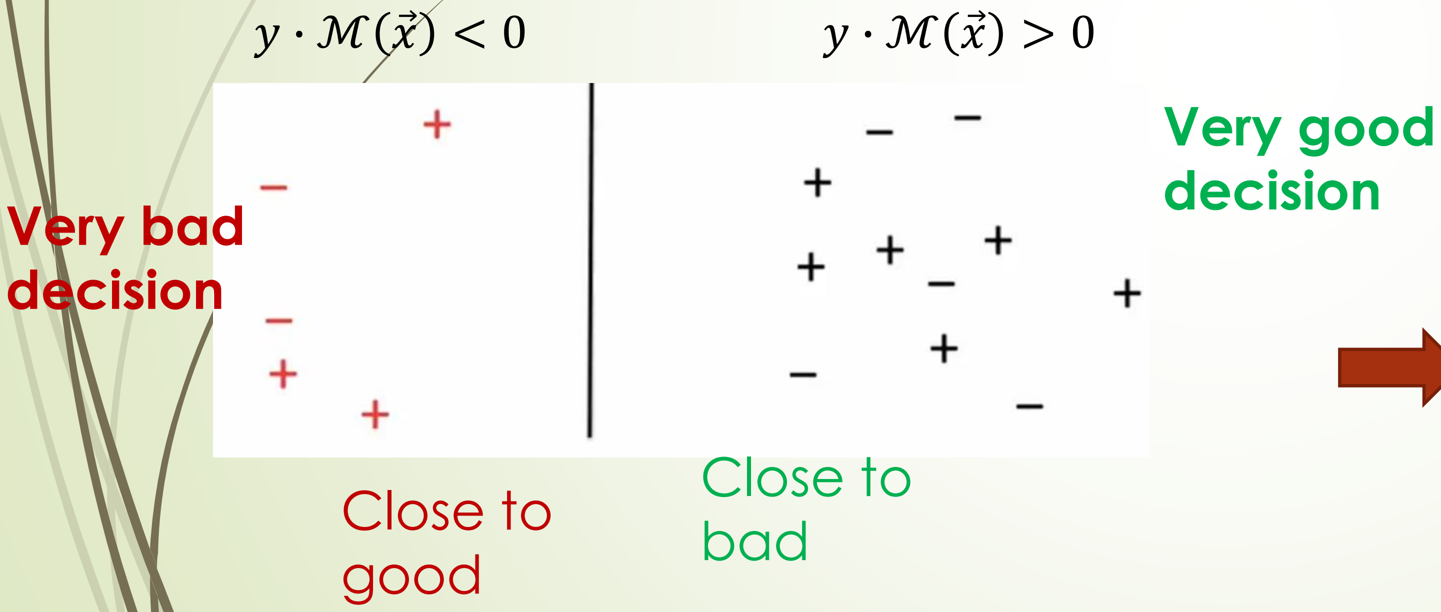
$$\mathcal{L} = \frac{1}{n}\sum_i 1_{[y \cdot \mathcal{M}(\vec{x}_i) < 0]}$$



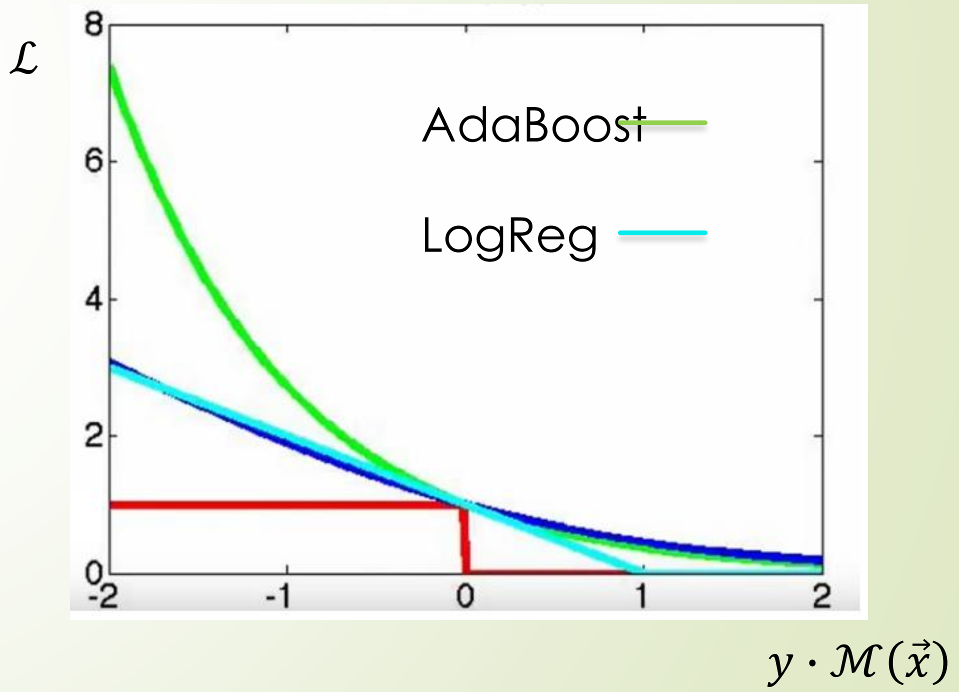
Max penalty each time!

Loss function (III)

- In theory such loss function is very powerfull, but in practice we cannot optimise such expression in any easy way and on top of this it has so sensitivity on how bad the decision was, i.e., each time the penalty is maximal



$y \cdot \mathcal{M}(\vec{x}) < 0$
$y \cdot \mathcal{M}(\vec{x}) > 0$
Very bad decision
Very good decision
Close to good
Close to bad



Loss function (IV)

- There are some tantalising facts regarding the loss function: the whole training process depends on the way we measure its performance – more aggressive approach may be more beneficial, it may determine how long the training process take and if it will be successful at all – **how interesting**
- Different loss functions determines upper limits w.r.t $1_{[y\cdot\mathcal{M}(\vec{x}_i)<0]}$ one:

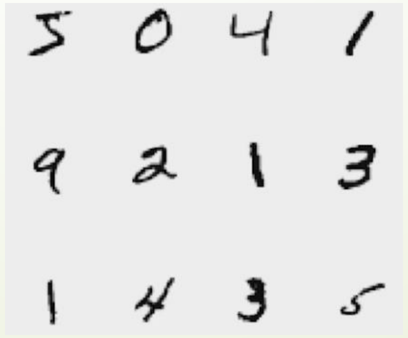
$$\mathcal{L}(y_i,\mathcal{M}(\vec{x}_i)) = \frac{1}{n}\sum_i [y_i \neq sign(\mathcal{M}(\vec{x}_i))] = \frac{1}{n}\sum_i 1_{[y\cdot\mathcal{M}(\vec{x}_i)<0]} \leq \frac{1}{n}\sum_i f_{\mathcal{M}}(y\cdot\mathcal{M}(\vec{x}_i))$$

- Our main target then should be:

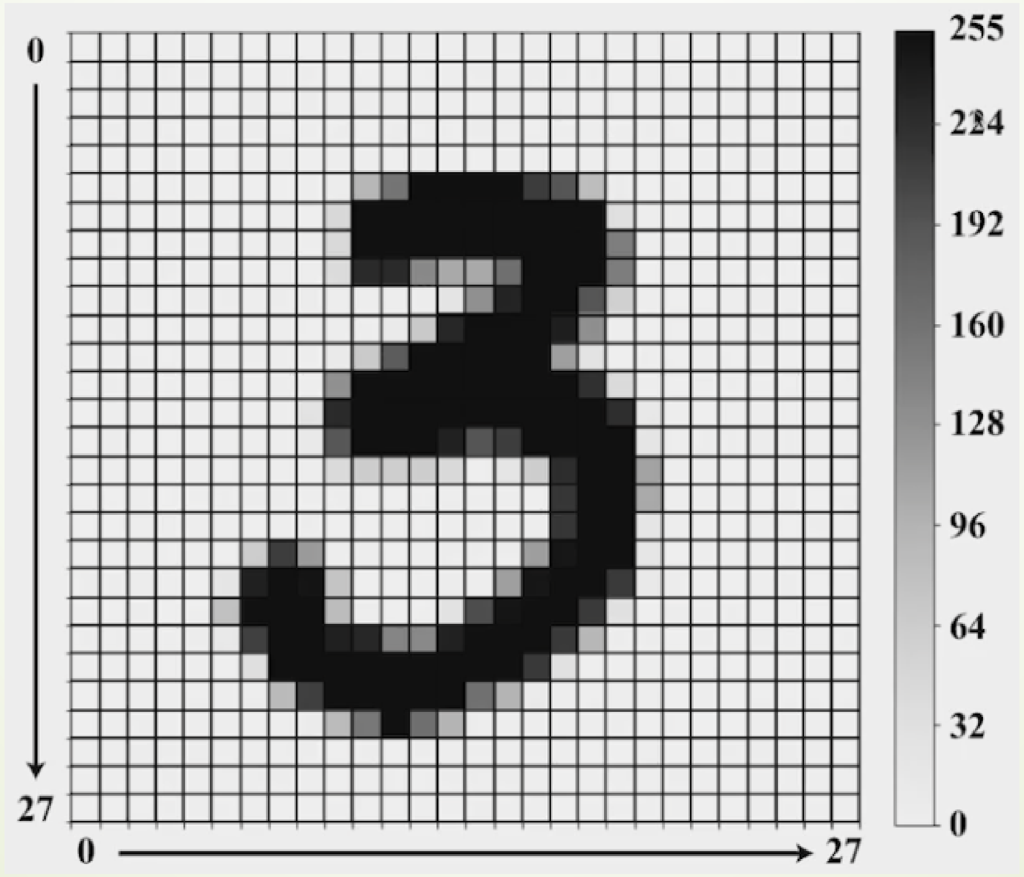
LAB classes

The most crucial point

- ***Data, data and data***
- Story of **LeNET** – model (successful deployment as automated system for zip code recognition for US postal services)
- Now, it is know as **MNIST** data set (~60 000 digits and ~10 000 for validation)
- An excellent example of almost perfect data set
  - Not too big – problem with processing and storing
  - The handwritten examples quite challenging for ML algorithms
  - Sufficiently diverse to allow for generalisation

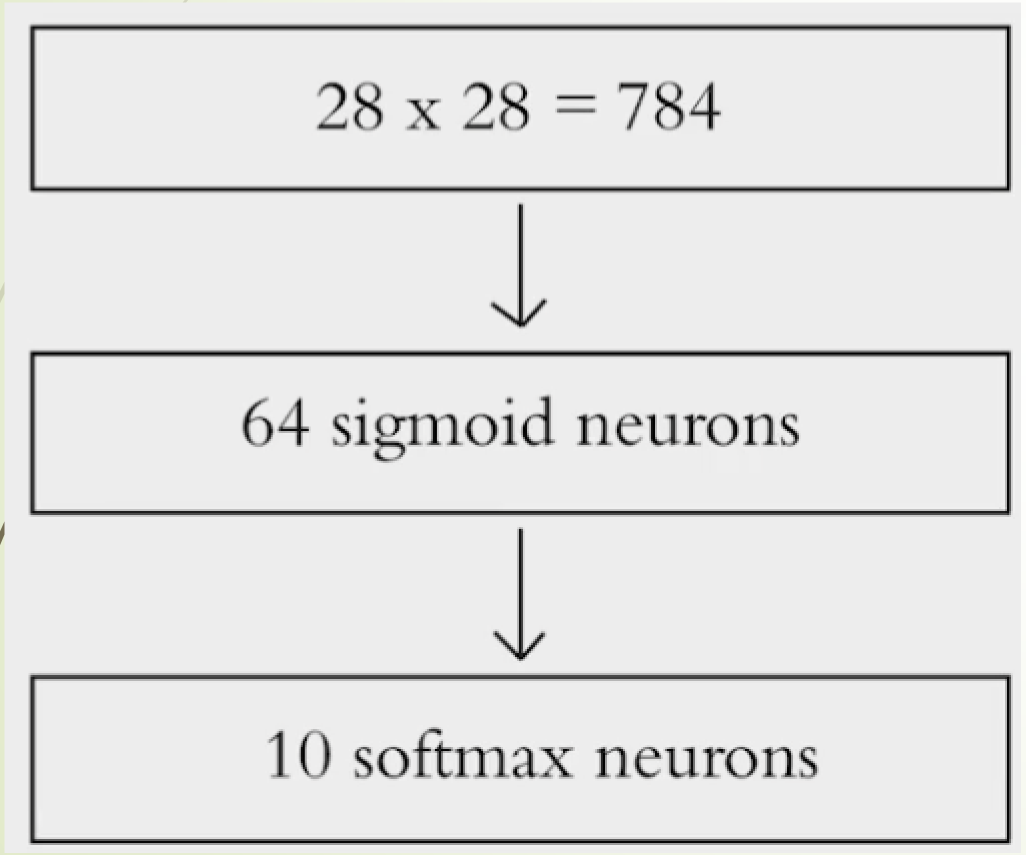


The most crucial point



- ❑ 28 x 28 pixels
- ❑ With 8 bit depth

Into the battle!

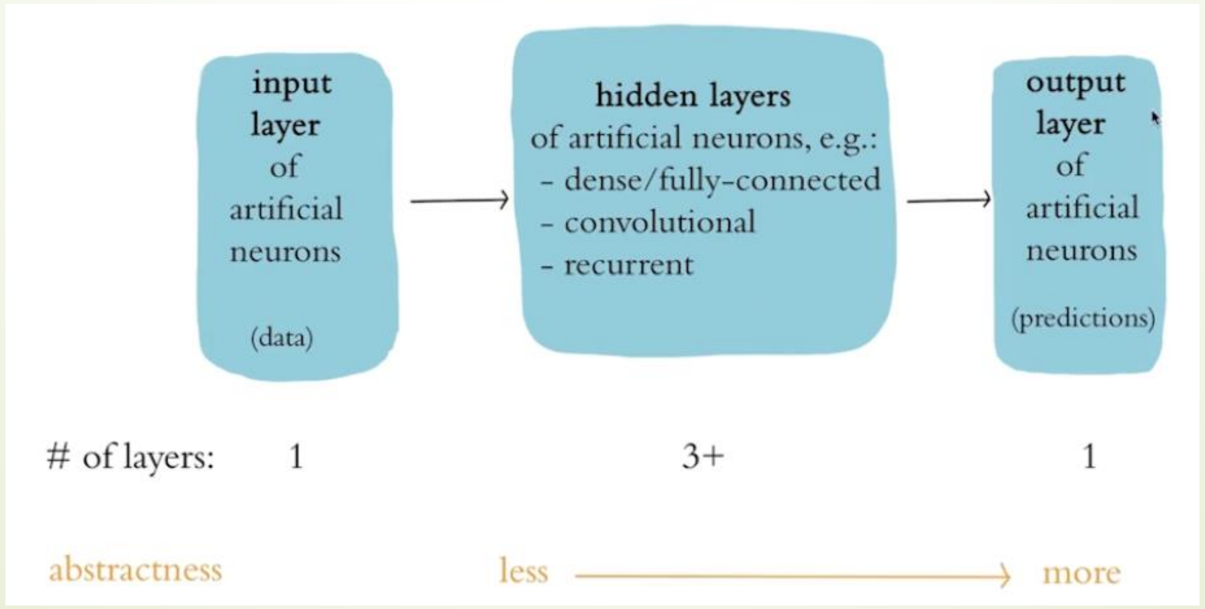


- ❑ Building the network architecture
- ❑ Input layer must be **related to the input data**
- ❑ Here, ***we collapse the input into a one dimensional structure***!
- ❑ Digits are quite simple geometric shapes, thus, we use small number of neurons
- ❑ The output layer is called **softmax** and represents the **number of classes** in our prediction

- ❑ It is not a deep model, but this shallow one will suffice!

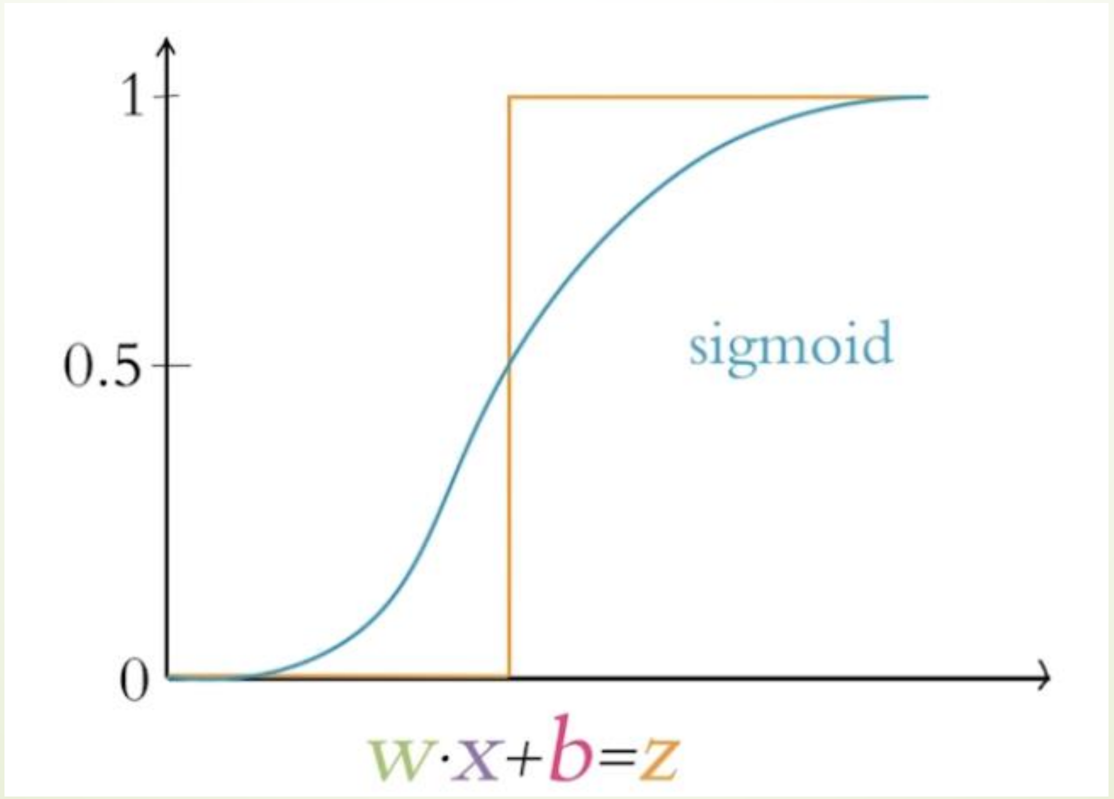
Building deep model

- ❑ ***Based on the solution we found during the lab classes build a deep model***
- ❑ **Add at least 2 additional layers**
- ❑ **Play with the number of neurons in the layers**
- ❑ **Keep the layers as dense for the time being**


input layer of artificial neurons
(data)
hidden layers of artificial neurons, e.g.:
- dense/fully-connected
- convolutional
- recurrent
output layer of artificial neurons
(predictions)
# of layers:
1
3+
1
abstractness
less
more

Sigmoind perceptron

- ***Alanyse the properties of sigmoid, ReLU and tanh curves***
- ***Explore Keras library regarding the activation functions***



The end